ACSAR: Software Model Checking with Transfinite Refinement

ACSAR: Software Model Checking with Transfinite Refinement

Mohamed Nassim Seghir and Andreas Podelski

Universität Freiburg

1 Introduction

ACSAR (Automatic Checker of Safety properties based on Abstraction Refinement) is a software model checker for C programs in the spirit of Blast [5], F-Soft [6], Magic [4] and Slam [1]. It is based on the counterexample-guided abstraction refinement (CEGAR) paradigm. Its specificity lies in the way it overcomes a problem common to all tools based on this paradigm. The problem arises from creating more and more spurious counterexamples by unfolding the same (`while`- or `for`-) loop over and over again; this leads to an infinite or at least too large sequence of refinement steps. The idea behind ACSAR is to abstract not just states but also the state changes induced by structured program statements, including for- and while-statements. The use of the new abstraction allows one to shortcut such a "transfinite" sequence of refinement steps.

The divergence of the abstraction refinement loop is not just a theoretical problem but one that hits us in our practical use of software model checker. ACSAR is integrated in a higher order theorem prover, namely Isabelle [3]. It is called, from within Isabelle, for discharging automatically generated verification obligations. Thus, another specificity of ACSAR as a software model checker lies in the way that it is used. We report on our experience of using ACSAR at the end of the paper.

2 A motivating example

Let us illustrate the need of abstracting loops through the example in Figure 1(a). This example is taken from the list of benchmarks that were used by McMillan and Jhala [7]. It represents the concatenation of two strings. The key word *assume* does not exist in the C language but it is used for the model checker to express additional assumptions.

A classical refinement generates predicates $i \geq 200, j < 100, i + 1 \geq 200, j + 1 < 100, i + 2 \geq 200, j + 2 < 100 \ldots i + 99 \geq 200, j + 99 < 100$. The loop is unrolled as many times as the number of loop iterations in a real execution. Moreover, if we want to perform a generic verification for arbitrary string length, by substituting $size$ for 100 in line 12 and 21, and $size * 2$ for 200 in line 25, the refinement process completely diverges. The problem is inherent to the CEGAR scheme in its present form (based on state abstraction) where the loop (15, 16, 17, 18, 19, 15) is unfolded over and over again. In this case, neither the interpolation approach nor the split prover method seem to help [7].

```
1  main(){
2
3   char x[101], y[101], z[201];
4    int i,j,k;
5
6    i = 0;
7    while(x[i] != 0){
8      z[i] = x[i];
9      i++;
10   }
11   /* length of x is less than 100 */
12   assume(i < 100);
13
14   j = 0;
15   while(y[j] != 0){
16     z[i] = y[j];
17     i++;
18     j++;
19   }
20   /* length of y is less than 100 */
21   assume(j < 100);
22
23   z[j] = 0;
24   /* prove we don't overflow z */
25   if(i >= 200)
26     {ERROR: goto ERROR;}
27 }
```

(a)

```
1  main(){
2
3   char x[101], y[101], z[201];
4    int i,j,k;
5
6    i = 0;
7    while(x[i] != 0){
8      z[i] = x[i];
9      i++;
10   }
11   /* length of x is less than 100 */
12   assume(i < 100);
13
14   j = 0;
15   if(*){
16     assume((j_next - j)==(i_next - i))
17     i = i_next;
18     j = j_next;
19   }
20   /* length of y is less than 100 */
21   assume(j < 100);
22
23   z[j] = 0;
24   /* prove we don't overflow z */
25   if(i >= 200)
26     {ERROR: goto ERROR;}
27 }
```

(b)

Fig. 1. Example in C code before and after the abstraction

3 Loop abstraction approach

As alternative to the iterative unfolding of loops, ACSAR approximates state changes induced by the execution of the loop. The idea of abstracting transitions was previously used to prove the termination of programs [2]. Our use of transition abstraction is in the context of checking safety properties.

3.1 How does ACSAR abstract loops ?

ACSAR extracts the list of transition constraints corresponding to the program. Bellow is the transition constraints corresponding to the loop (15, 16, 17, 18, 19, 15) from the example in Figure 1(a).

$$pc = 15 \wedge y[j] \neq 0 \wedge z'[i] = y[j] \wedge i' = i + 1 \wedge j' = j + 1 \wedge pc' = 15 \quad (1)$$
$$pc = 15 \wedge y[j] = 0 \wedge pc' = 21 \quad (2)$$

A transition constraint is a conjunction of atomic formulas, it expresses a binary relation between a starting state and an arrival state of the program. In atomic formulas, variables marked with primes are evaluated in the arrival state of the transition, otherwise they are evaluated in the starting state. The special variable *pc* represents the program counter. When an atomic formula does not contain any variable marked with a prime it is called a *guard*. An atomic formula that contains variables with primes is called an *update*. If a variable does not appear marked with a prime in any atomic formula, then it means implicitly that its value does not change when the transition is performed.

The next step is the abstraction phase. In this phase non relevant guards are removed and constraints expressing relations between old and new values of the variables are extracted. For example: the expression $(i' - i) = (j' - j)$ is automatically extracted by ACSAR as both variables i and j increase by the same constant number within the loop. Transition constraints (1) and (2) are replaced by their abstractions (1') resp. (2'). To the difference of transition (1) its abstraction (1') does not loop but it approximates the effect of the loop over the program variables. With this abstraction ACSAR succeeds to prove that the program is safe.

$$pc = 15 \wedge i' - i = j' - j \wedge pc' = 21 \qquad (1')$$
$$pc = 15 \wedge pc' = 21 \qquad (2')$$

Question: How can one express the above abstraction of a loop in terms of a source-to-source transformation on the C program? The problem is that a transition constraint expresses a constraint on the after-value of a transition, but a program statement defines the after-value by the value of an expression. As often, the solution is very simple. We write the transition constraint as a program expression (using an *uninitialized* auxiliary variable `x_next` for the primed version of the variable `x`) and use the program expression in an `assume` statement and then add assignment statements of the form `x = x_next`. See Figure 1(b). The loop (15, 16, 17, 18, 19, 15) is replaced by a nondeterministic 'if' block (the 'nondeterministic' expression is denoted `*`).

What do we gain with loop abstraction? The benefit is two folds:

- We obtain better performance in terms of time and space. Table 1 illustrates a comparison between the loop abstraction approach and a simple approach based on the weakest precondition for refinement. We apply both approaches on different instances of the example of figure 1(a). Column *size* contains different values of the size of input array variables `x` and `y`. Implicitly, the size of `z` is $2 * size$. Using the simple approach, we clearly notice a nonlinear increase of the verification time in function of instance size. With the loop abstraction approach, the execution time is the same and relatively small for all the instances.
- Using the loop abstraction approach, we can verify a generic version (section 2) of the previous example. The abstract transition represents a *parameterization* of all paths corresponding to loop unfolding of different instances of the example program.

instance	size	time		number of states	
		simple	loop abstraction	simple	loop abstraction
1	10	1.19	0.29	12	5
2	20	2.77	/	22	/
3	50	33.59	/	52	/
4	75	127.72	/	77	/
5	100	336.56	/	102	/

Table 1. Performance comparison between the loop abstraction approach and the simple approach

4 ACSAR in short

ACSAR has the usual ingredients of a software model checker. It receives as input a C file consisting of functions and data structures. Location labels are used to specify a monitor for the property that we want to check. A global control-flow graph is obtained by inlining function bodies into the corresponding call sites. ACSAR translates the program into a set of transition constraints, its canonical representation. The main kernel of ACSAR is composed of two parts: the search engine that explores the state space (building the abstraction on the fly) and the counter example analyzer which increases the precision of the search engine when the abstraction is too coarse. For building the abstraction and, respectively, for checking consistency of transitions, both parts interact with a parameterized constraint solver such as Simplify. ACSAR builds the abstraction of loops on demand, namely when the counter example analyzer has detected that a loop has been unfolded twice. The threshold for the number of unfoldings is a parameter which, for now, is set to two.

5 Experimental Evaluation

ACSAR is used in the Verisoft project[1] as a back-end for the higher order interactive theorem prover Isabelle [3]. Isabelle has a Hoare logic module for the specification and verification of programs [9]. For a Hoare triple $P\ c\ Q$ Isabelle performs the proof of the postcondition Q in three steps: the proof that Q holds, the proof that the program c terminates and the proof that no run time errors occur during the execution of c under the precondition P. For this last step Isabelle generates proof obligations expressing necessary conditions for a safe execution of any command in the program c. For example, given the integer variable x and the command $x = x + 1$, Isabelle generates the proof obligation $MAXint \leq x \leq MINint$. The task of verifying such a proof obligation is automatically delegated to ACSAR. The overall goal is to minimize the 'manual'

[1] http://www.verisoft.de

interaction between the verification engineer and Isabelle. In the (ongoing) interactive verification effort for the Vamos micro-kernel (which is being developed within the Verisoft project), ACSAR automatically discharges about 75% of the (automatically generated) verification obligations (the remaining 25% concern properties that require variable quantification).

Outlook: We are planning to carry over methods for the generation of linear invariants [8] to our approach for abstracting loops. We want also to handle simple array assertions that involve quantifiers; e.g., $\forall i\ (0 \leq i < n) \Rightarrow a[i] = 0$.

References

1. Thomas Ball and Sriram K. Rajamani. The Slam project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

2. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In *SAS*, pages 87–101, 2005.

3. Matthias Daum, Stefan Maus, Norbert Schirmer, and M. Nassim Seghir. Integration of a software model checker into Isabelle. In *LPAR*, pages 381–395, 2005.

4. Sagar Chaki et al. Modular verification of software components in C. In *ICSE*, pages 385–395, 2003.

5. Thomas A. Henzinger et al. Software verification with BLAST. In *SPIN*, pages 235–239, 2003.

6. Franjo Ivancic, Ilya Shlyakhter, Aarti Gupta, and Malay K. Ganai. Model checking c programs using F-soft. In *ICCD*, pages 297–308, 2005.

7. Ranjit Jhala and Kenneth L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, pages 459–473, 2006.

8. Michael Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

9. Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In *LPAR*, pages 398–414, 2004.

APPENDIX

A ACSAR the tool

A.1 Environment

ACSAR is written in Gnu C++ under the Linux operating system. It was tested with success under the following versions of Linux: Debian, Suse and Gentoo.

A.2 Availability

A package containing the binary of ACSAR can be downloaded from the following address: http://www.mpi-inf.mpg.de/∼seghir/ACSAR/ACSAR-web-page.html.

A.3 Application

ACSAR is used in the Verisoft project as a back-end for the higher order interactive theorem prover Isabelle. Its role is to discharge runtime error guards, mainly for: overflows and underflows, array out of bounds and NULL pointer dereferences. Among the applications that we verified: the Vamos micro-kernel. We are actually verifying a string library and the C_0 compiler(C_0 is a subset of the C language). 25% of the functions constituting the C_0 compiler were already verified. Figure 2 shows program representation within the Isabelle proof environment. Guards are given between '{' '}'. The result of the verification after calling ACSAR is shown in Figure 3. Figure 4 shows the case when a guard does not hold, in this situation a counter example is generated.

B Demo

In this section we describe our demonstration plan. First, we consider example of Figure 5 to show how to specify a property and verify whether it holds using ACSAR. This example represents a routine from Linux-2.4.0 CD-ROM interface device driver. For readability concerns, portions of the code that are not relevant to the property that we want to check are omitted.

As verifier, one does not require to understand the functionality of the code to be verified, however, one should know what to verify and how to specify it. Fortunately, for our example of Figure 5, the developer documented his code by adding the following comment:

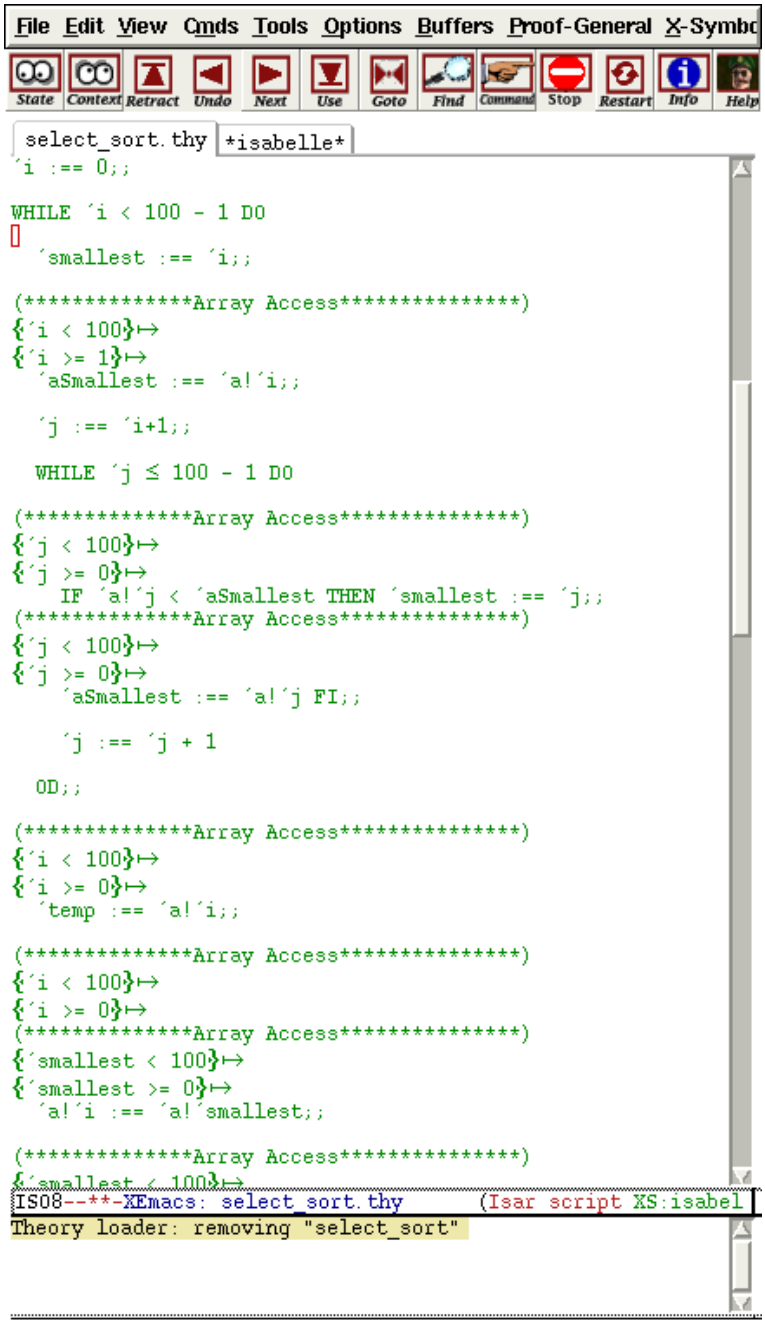

Fig. 2. Program representation in Isabelle

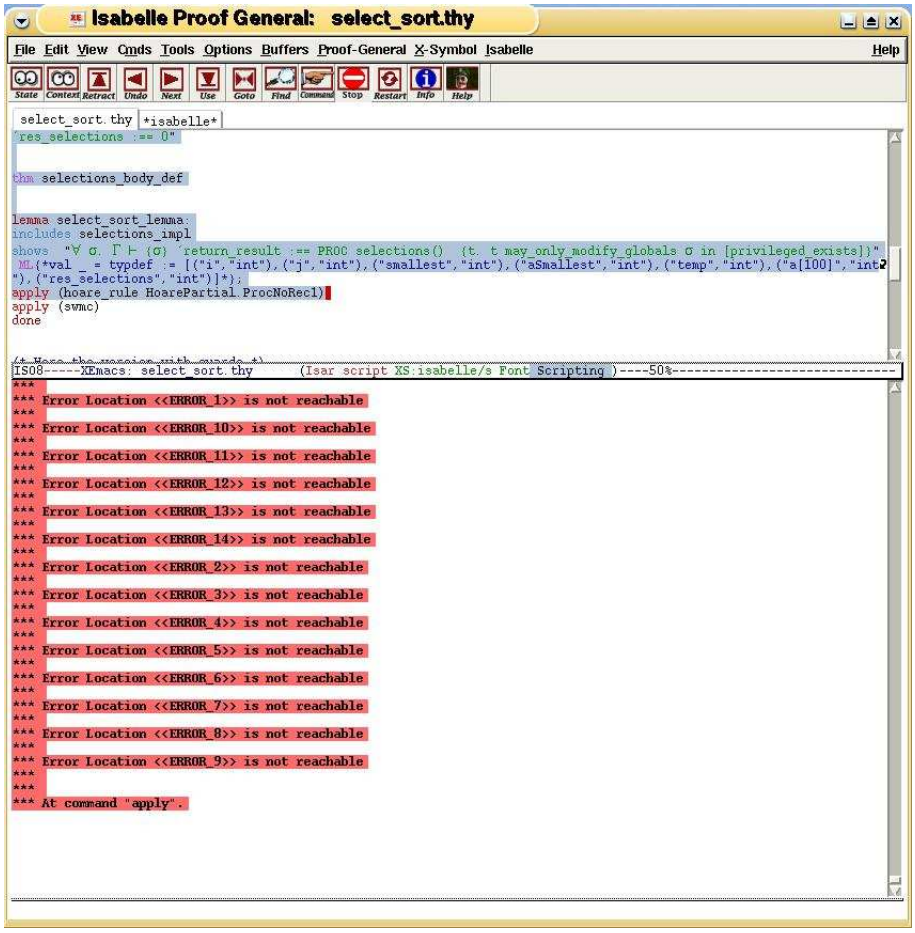

Fig. 3. Result of Isabelle after calling ACSAR

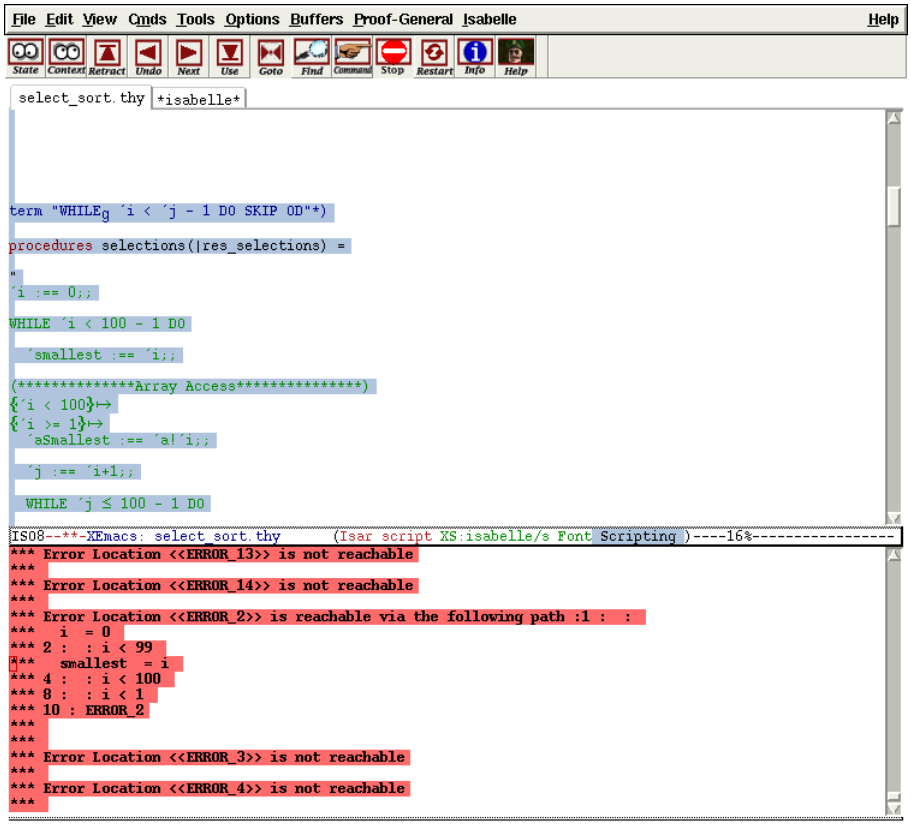

Fig. 4. Case of counter example

```
The result register can be read 10 bytes at a time, a wait for
  result to be asserted must be done between every 10 bytes
```

This constitutes for us an informal description of a behavioral property that the implementation should fulfill. A formalization of this assertion is required so that we can check it with ACSAR. For this, we use functions called by the routine as stubs and exploit them to define a monitor for the property to be checked. The function `read_result_register` is the one that reads the result register and `is_result_ready` is the one that is called to wait for the result. The specification can then be stated as follows: between every ten calls of function `read_result_register` we must call the function `is_result_ready` at least once. We introduce the variable `monitor`, it is incremented each time the function `read_result_register` is called, this is illustrated in Figure 6. If `monitor` becomes greater than ten then error label `ERROR_1` is reachable, meaning that the specification is violated. When `is_result_ready` is called, `monitor` is set to zero.

We call acsar with the command:

```
acsar --reach --mainproc get_result --file linux_cd_driver.c
```

Option '`reach`' specifies that we want to check reachability, '`mainproc`' specifies the root procedure and '`file`' refers to the source file of the program that we want to verify. As result, ACSAR returns a counter example. See Figure 7. Local variables are renamed by concatenating their names with the name of the function in which they are declared. Numbers at the beginning of lines represent *pc* internal values (not line numbers in the source file). The counter example says that variable `monitor` becomes greater than ten after it is incremented once and we jump to ERROR_1. How can this happen? In fact, this scenario is possible as `monitor` was not initialized before the call to `read_result_register` at line 9 in Figure 5. Thus, after this call `monitor` can have any value. By initializing `monitor` to zero before line 9, ACSAR answers that the program is safe.

B.1 Example for loop abstraction

Now we present a case where loop abstraction is required. We take the example of Figure 1(a), previously presented in this paper. By calling ACSAR on that example we notice that the verification process loops without terminating. Let us now call ACSAR with the command:

```
acsar --reach --mainproc main --file string_concat1.c --loopsumr
```

The '`loopsumr`' option tells ACSAR to abstract a loop if it is unfolded more than twice during the verification. This time ACSAR is able to prove the safety of the program. To see the transformation occurring to the program, we use the option '`printloopsumr`' that displays transition constraints constituting the loop and the abstraction of the loop as a single transition constraint. See Figure 8. `pc_1` represents the value of `pc` after performing the transition. Like in the C language '&&' expresses the logical AND. The conjunct appearing before

the occurrence of `pc_1` represents the guard and the one appearing after the occurrence of `pc_1` represents the update. We notice in the abstraction that new variables `temp_var_1` and `temp_var_2` are introduced. They correspond respectively to the values of `i` and `j` after executing the loop. Transition constraints constituting the original loop are disabled.

```
1  static void
2  get_result (unsigned char *result_buffer,
3              unsigned int *result_size)
4  {
5    unsigned char a, b;
6    int i, res;
7    unsigned int retry_count;
8
9    b = read_result_register ();
10   if(ERROR == 1) goto ERROR_1;
11   if ((a & 0xf0) != 0x20)
12   {
13     if (b > 8)
14     {
15       for (i=0; i<8; i++)
16       {
17         * result_buffer = read_result_register ();
18         if(ERROR == 1) goto ERROR_1;
19         // .....
20       }
21       b = b - 8;
22       while (b > 10)
23       {
24         res = is_result_ready ();
25         while ((retry_count > 0) && (!res))
26         {
27           res = is_result_ready ();
28           // .....
29         }
30         res = is_result_ready ();
31
32         // .....
33
34         for (i=0; i<10; i++)
35         {
36           * result_buffer = read_result_register ();
37           if(ERROR == 1) goto ERROR_1;
38         }
39         b = b - 10;
40       }
41       if (b > 0)
42       {
43         res = is_result_ready ();
44         // .....
45       }
46     }
47     while (b > 0)
48     {
49       * result_buffer = read_result_register ();
50       if(ERROR == 1) goto ERROR_1;
51       b--;
52     }
53   }
54   goto end;
55   ERROR_1:;
56   end:;
57 }
```

Fig. 5. CD-ROM driver routine

```
1   int is_result_ready ()
2   {
3     monitor = 0;
4     if(ACSAR_NONDET)
5     {
6       return 0;
7     }
8     else
9     {
10      return 1;
11    }
12  }
13
14   read_result_register ()
15  {
16    ++ monitor;
17    if(monitor > 10)
18    ERROR = 1;
19    else ERROR = 0;
20  }
```

Fig. 6. Monitor corresponding to the CD-ROM driver

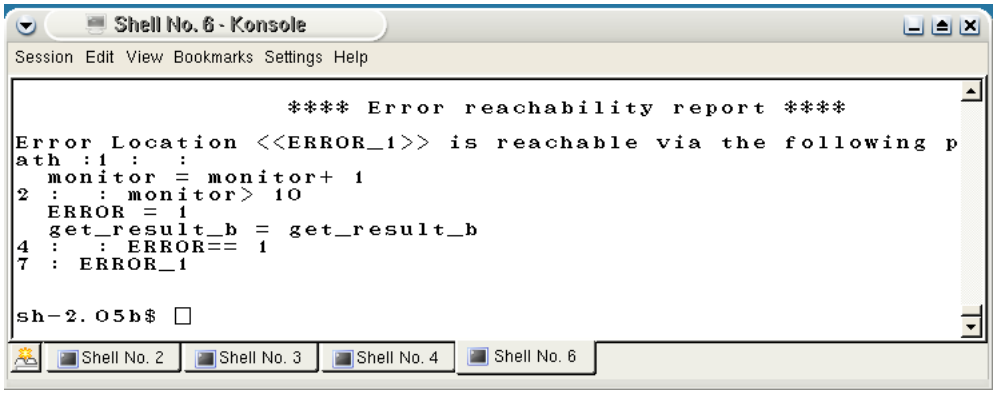

Fig. 7. Counter example generated by ACSAR

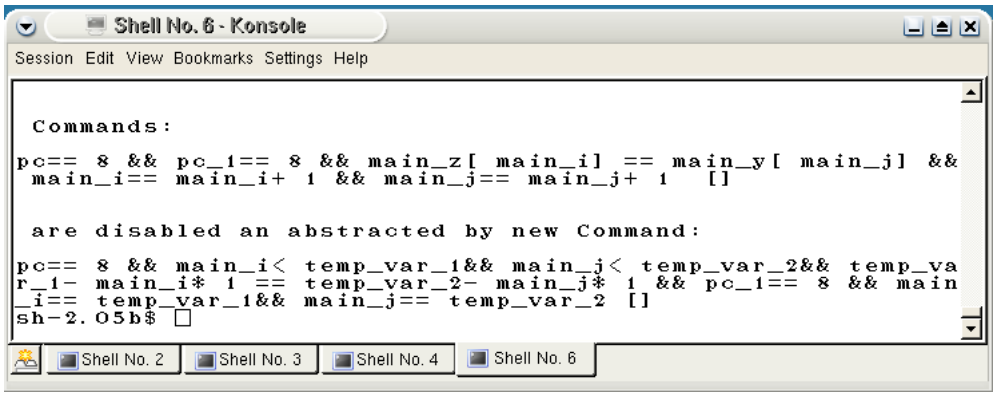

Fig. 8. Loop abstraction computed by ACSAR